Parking Lot Design C++

Lld of parking lot, in cpp, give all details and requirements as well, show diagrams if required, or design patterns which may be used.

Reasoned about parking lot design in C++ for 11 seconds

Below is an example of how you might design a Parking Lot system in C++ using object-oriented principles. This example covers:

1. **Requirements and Assumptions**
2. **Class Design and Relationships (with Diagrams)**
3. **Design Patterns Used**
4. **Example C++ Code**

---

1. Requirements and Assumptions

Functional Requirements

- **Vehicle Types:** The system supports multiple vehicle types (e.g., **Motorcycle**, **Car**, **Bus**).
- **Parking Spot Types:** Spots are designed for different vehicle sizes (e.g., **Motorcycle**, **Compact**, **Large**).
  - A **Motorcycle** can park in any spot.
  - A **Car** can park in **Compact** or **Large** spots.
  - A **Bus** requires several contiguous **Large** spots.
- **Operations:**
  - **Park a Vehicle:** Find an appropriate spot (or spots, in the case of a bus) and park the vehicle.
  - **Unpark a Vehicle:** Free up the spot(s) once the vehicle leaves.
  - **Display Availability:** List free spots by type or level.

Non-functional Requirements

- **Modularity and Maintainability:** The design should follow SOLID principles.
- **Scalability:** Easy to add more levels, spot types, or vehicle types.
- **Reusability:** Use design patterns (e.g., **Factory**, **Strategy**, **Facade**) to decouple components.

---

2. Class Design and Relationships

Key Classes

- **ParkingLot:** The top-level class that contains multiple **Level** objects. Acts as a **Facade** for parking/unparking operations.
- **Level:** Represents a single floor in the parking lot and contains many **ParkingSpot** objects.
- **ParkingSpot:** Represents an individual parking spot. It holds its size/type and the parked vehicle (if any).

- **Vehicle (Abstract Base Class):** Base class for all vehicles with derived classes like **Motorcycle**, **Car**, and **Bus**.

Class Diagram

Below is a simplified UML-like diagram illustrating the relationships:

```
plaintext


          +-------------------+
          |    ParkingLot     |
          +-------------------+
          | - levels: vector<Level>   |
          +-------------------+
          | + parkVehicle(Vehicle*)   |
          | + unparkVehicle(Vehicle*) |
          +-------------------+
                    |
                    |
          +-------------------+
          |      Level      |
          +-------------------+
          | - spots: vector<ParkingSpot> |
          | - levelNumber: int            |
          +-------------------+
          | + parkVehicle(Vehicle*)       |
          +-------------------+
                    |
                    |
          +-------------------+
          |   ParkingSpot     |
          +-------------------+
          | - spotType: VehicleSize       |
          | - isOccupied: bool            |
          | - parkedVehicle: Vehicle*     |
          +-------------------+
          | + canFitVehicle(Vehicle*)     |
          | + park(Vehicle*)              |
          | + unpark()                    |
          +-------------------+

          +-------------------+
          |     Vehicle       |<--- Abstract Class
          +-------------------+
          | - licensePlate: string        |
          +-------------------+
          | + getSize() : VehicleSize     |
          +-------------------+
                 /   |   \
                /    |    \
     +---------+   +---------+   +---------+
     |Motorcycle|   |   Car   |   |   Bus   |
     +---------+   +---------+   +---------+
```

Note: `**VehicleSize**` is an enumeration representing different spot/vehicle sizes.

---

3. Design Patterns Used

- **Facade Pattern:**
  - The `**ParkingLot**` class provides a simple interface to park and unpark vehicles, hiding the internal complexity of levels and spots.
- **Factory Pattern:**
  - When creating vehicles (or even parking spots), a factory can encapsulate object creation details. (For brevity, the code sample below directly instantiates the objects.)

- **Strategy Pattern (Optional):**
  - You can encapsulate parking spot selection algorithms (for example, different strategies for parking a bus vs. a car) into separate strategy classes.

---

4. Example C++ Code

Below is an example implementation. In a production system, you'd further modularize, add error handling, and consider thread safety.

```
cpp
```

```cpp
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

// Enumeration for vehicle and spot sizes
enum class VehicleSize { Motorcycle, Compact, Large };

// ----------------------
// Vehicle Base Class
// ----------------------
class Vehicle {
public:
    Vehicle(const string& plate) : licensePlate(plate) {}
    virtual ~Vehicle() {}

    string getLicensePlate() const { return licensePlate; }
    virtual VehicleSize getSize() const = 0;

protected:
    string licensePlate;
};

class Motorcycle : public Vehicle {
public:
    Motorcycle(const string& plate) : Vehicle(plate) {}
    VehicleSize getSize() const override {
        return VehicleSize::Motorcycle;
    }
};

class Car : public Vehicle {
public:
    Car(const string& plate) : Vehicle(plate) {}
    VehicleSize getSize() const override {
        return VehicleSize::Compact;
    }
};

class Bus : public Vehicle {
public:
    Bus(const string& plate) : Vehicle(plate) {}
    // For a bus, we consider its "size" as Large
    VehicleSize getSize() const override {
        return VehicleSize::Large;
    }
};

// ----------------------
// ParkingSpot Class
// ----------------------
class ParkingSpot {
public:
    ParkingSpot(VehicleSize spotSize)
        : spotSize(spotSize), occupied(false), parkedVehicle(nullptr) {}

    bool isAvailable() const { return !occupied; }
```

```
    // Determines if the spot can fit the vehicle based on size
    bool canFitVehicle(const Vehicle& vehicle) const {
        // Motorcycle can fit anywhere.
        if (vehicle.getSize() == VehicleSize::Motorcycle)
            return true;

        // Car can park in Compact or Large spots.
        if (vehicle.getSize() == VehicleSize::Compact) {
            return spotSize == VehicleSize::Compact || spotSize == VehicleSize::Large;
        }

        // Bus requires a Large spot.
        if (vehicle.getSize() == VehicleSize::Large) {
            return spotSize == VehicleSize::Large;
        }
        return false;
    }

    bool park(Vehicle* v) {
        if (!occupied && canFitVehicle(*v)) {
            parkedVehicle = v;
            occupied = true;
            return true;
        }
        return false;
    }

    void unpark() {
        parkedVehicle = nullptr;
        occupied = false;
    }

    void displayStatus() const {
        cout << "Spot [" << (spotSize == VehicleSize::Motorcycle ? "Motorcycle" :
                             spotSize == VehicleSize::Compact ? "Compact" : "Large")
             << "] - " << (occupied ? "Occupied" : "Free") << endl;
    }

private:
    VehicleSize spotSize;
    bool occupied;
    Vehicle* parkedVehicle;
};

// -----------------------
// Level Class
// -----------------------
class Level {
public:
    Level(int levelNumber, int numSpots)
        : levelNumber(levelNumber)
    {
        // For simplicity, we create spots in fixed proportions:
        // 20% Motorcycle, 60% Compact, 20% Large
        int motorcycleSpots = numSpots / 5;
        int compactSpots = numSpots * 3 / 5;
        int largeSpots = numSpots - (motorcycleSpots + compactSpots);

        for (int i = 0; i < motorcycleSpots; i++) {
            spots.emplace_back(VehicleSize::Motorcycle);
        }
        for (int i = 0; i < compactSpots; i++) {
            spots.emplace_back(VehicleSize::Compact);
        }
        for (int i = 0; i < largeSpots; i++) {
            spots.emplace_back(VehicleSize::Large);
        }
    }

    // Attempt to park the vehicle in one of the available spots
    bool parkVehicle(Vehicle* vehicle) {
        // For Bus, you might need to check for multiple contiguous spots.
        // Here, we assume non-bus vehicles.
        for (auto& spot : spots) {
```

```
            if (spot.isAvailable() && spot.canFitVehicle(*vehicle)) {
                if (spot.park(vehicle)) {
                    cout << "Vehicle " << vehicle->getLicensePlate()
                         << " parked on Level " << levelNumber << endl;
                    return true;
                }
            }
        }
        return false;
    }

    void unparkVehicle(Vehicle* vehicle) {
        for (auto& spot : spots) {
            // For simplicity, we check the license plate.
            // In a real system, you'd track the spot or use a map.
            // Here, we simply free the first occupied spot.
            // (Additional logic is required to exactly identify the vehicle.)
            if (!spot.isAvailable()) {
                // We assume that the spot has the vehicle if parked.
                // In a full implementation, each spot would store the pointer.
                spot.unpark();
                cout << "Vehicle " << vehicle->getLicensePlate()
                     << " unparked from Level " << levelNumber << endl;
                return;
            }
        }
    }

    void displayStatus() const {
        cout << "Level " << levelNumber << " status:" << endl;
        for (const auto& spot : spots) {
            spot.displayStatus();
        }
    }

private:
    int levelNumber;
    vector<ParkingSpot> spots;
};

// -----------------------
// ParkingLot Class (Facade)
// -----------------------
class ParkingLot {
public:
    ParkingLot(int numLevels, int spotsPerLevel) {
        for (int i = 0; i < numLevels; i++) {
            levels.emplace_back(i, spotsPerLevel);
        }
    }

    bool parkVehicle(Vehicle* vehicle) {
        // Try each level until the vehicle is parked.
        for (auto& level : levels) {
            if (level.parkVehicle(vehicle)) {
                return true;
            }
        }
        cout << "No available spot for vehicle " << vehicle->getLicensePlate() << endl;
        return false;
    }

    void unparkVehicle(Vehicle* vehicle) {
        for (auto& level : levels) {
            level.unparkVehicle(vehicle);
        }
    }

    void displayStatus() const {
        for (const auto& level : levels) {
            level.displayStatus();
        }
    }

private:
```

```
    vector<Level> levels;
};

// -----------------------
// Main Function to Demonstrate Usage
// -----------------------
int main() {
    // Create a parking lot with 3 levels and 30 spots per level
    ParkingLot parkingLot(3, 30);

    // Create some vehicles
    Motorcycle moto("MOTO123");
    Car car("CAR456");
    Bus bus("BUS789");

    // Park vehicles
    parkingLot.parkVehicle(&moto);
    parkingLot.parkVehicle(&car);
    parkingLot.parkVehicle(&bus);

    // Display status of parking lot
    parkingLot.displayStatus();

    // Unpark a vehicle
    parkingLot.unparkVehicle(&car);

    cout << "\nAfter unparking the car:" << endl;
    parkingLot.displayStatus();

    return 0;
}
```

Summary

- **Requirements:** We covered vehicle and spot types along with basic operations.
- **Design:** The system is divided into multiple classes with clear responsibilities. The `**ParkingLot**` class acts as the main interface.
- **Patterns:** We highlighted the use of the **Facade**, **Factory**, and **Strategy** patterns (the latter can be further expanded).
- **Code Example:** The provided C++ code demonstrates how the classes interact to perform parking and unparking operations.

This design can be further expanded to include advanced features (like handling buses that require multiple contiguous spots, real-time availability tracking, or payment integration) while maintaining a clean separation of concerns.

Anything else which could be added or improved, or asked for software Engineer interviews

Thought for a second ›

Beyond the basic implementation, there are several enhancements and edge cases you can address to take your design further—especially in an interview setting. Here are some improvements and potential discussion points:

Additional Features and Improvements

- **Concurrency and Thread Safety:**
  Incorporate robust locking or concurrent data structures to handle simultaneous parking requests. You might use mutexes or even atomic operations to prevent race conditions when multiple threads are trying to update parking spot status.
- **Bus and Multi-Spot Vehicle Support:**
  Extend your design to handle vehicles that require multiple contiguous spots (e.g., buses). This might involve an algorithm to scan for a sequence of free spots, and you could discuss trade-offs in terms of efficiency.
- **Dynamic Pricing and Reservations:**
  Add a reservation system and dynamic pricing model that could adjust rates based on demand or time of day. This adds complexity but is a common real-world requirement and a great discussion point.
- **Improved Error Handling and Logging:**
  Ensure that all operations (e.g., parking, unparking, reservation cancellation) have clear error-handling and logging mechanisms to support maintainability and debugging in production environments.
- **Design Patterns:**
  - **Facade Pattern:** Further encapsulate subsystems like spot management, payment, and display updates behind a single interface.
  - **Observer Pattern:** Implement an observer to update display boards or notify attendants when the status of parking spots changes.
  - **Strategy Pattern:** Use it for parking spot allocation algorithms, allowing you to swap between different strategies (e.g., nearest-spot-first vs. space-optimization).
- **Scalability and Extensibility:**
  Discuss how the design can be extended to handle additional levels, new vehicle types, and integration with external services (like payment gateways or sensor networks).

Interview Discussion Points

- **Trade-offs in Data Structures:**
  Why did you choose a vector (or list) for storing spots? What would be the advantages of using a priority queue or a bitmap for faster availability checks?
- **Handling Edge Cases:**
  How do you handle scenarios when the parking lot is nearly full, or when a bus arrives and needs several contiguous spots? What are your strategies for failure or rollback if something goes wrong?
- **System Scaling:**
  How would your design adapt if the parking lot were to expand into a multi-level automated facility with thousands of spots? What changes might be needed in the parking spot manager?
- **Integration with External Systems:**
  How would you incorporate real-time sensor data or integrate with a payment system for automated fee collection? How does this affect the overall architecture?
- **Testing and Maintainability:**
  What unit or integration tests would you write to ensure the reliability of your parking management system? How do you make your system easy to maintain and extend?

Discussing these enhancements shows you understand not just the basic requirements, but also the broader challenges of designing robust, scalable, and maintainable systems. This depth of understanding is often what interviewers look for in software engineering candidates.